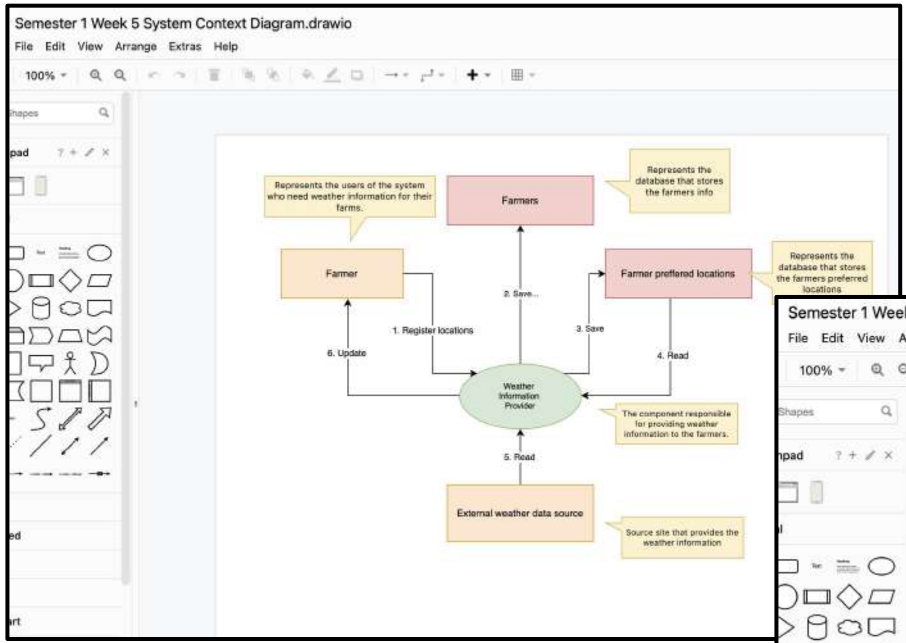
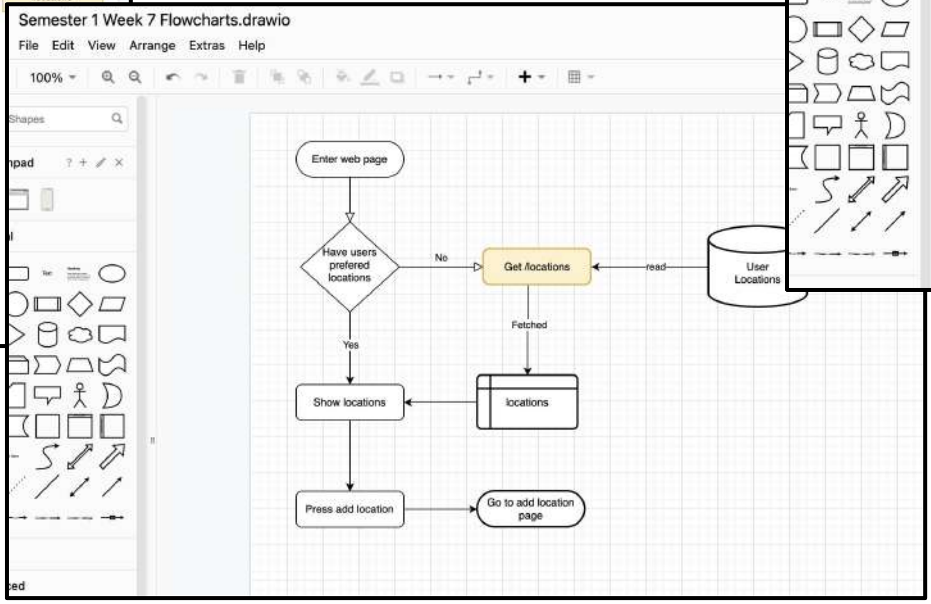
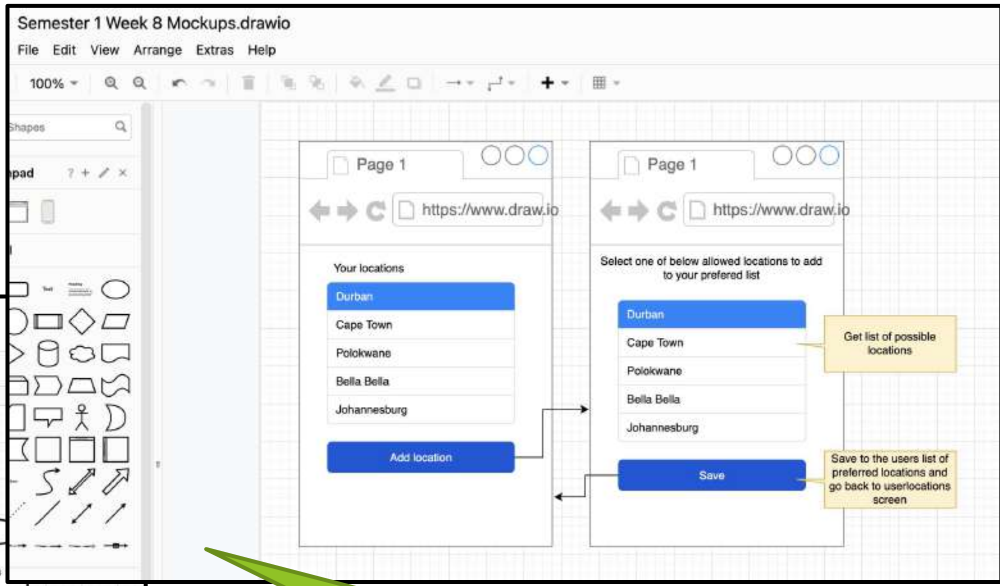
Semester 3

Week 6: Designing our screen components


Semester 1 Week 5 System Context Diagram.drawio
File Edit View Arrange Extras Help
100%
Represents the users of the system who need weather information for their farms.
Farmers
Represents the database that stores the farmers info
Farmer
Farmer preffered locations
Represents the database that stores the farmers preferred locations
2. Save...
1. Register locations
3. Save
6. Update
4. Read
Weather Information Provider
The component responsible for providing weather information to the farmers.
5. Read
External weather data source
Source site that provides the weather information




Semester 1 Week 8 Mockups.drawio
File Edit View Arrange Extras Help
100%
Page 1
https://www.draw.io
Your locations
Durban
Cape Town
Polokwane
Bella Bella
Johannesburg
Add location
Page 1
https://www.draw.io
Select one of below allowed locations to add to your prefered list
Durban
Cape Town
Polokwane
Bella Bella
Johannesburg
Get list of possible locations
Save
Save to the users list of preferred locations and go back to userlocations screen

You now need to design your screen components to suite your design specs created in semester 1

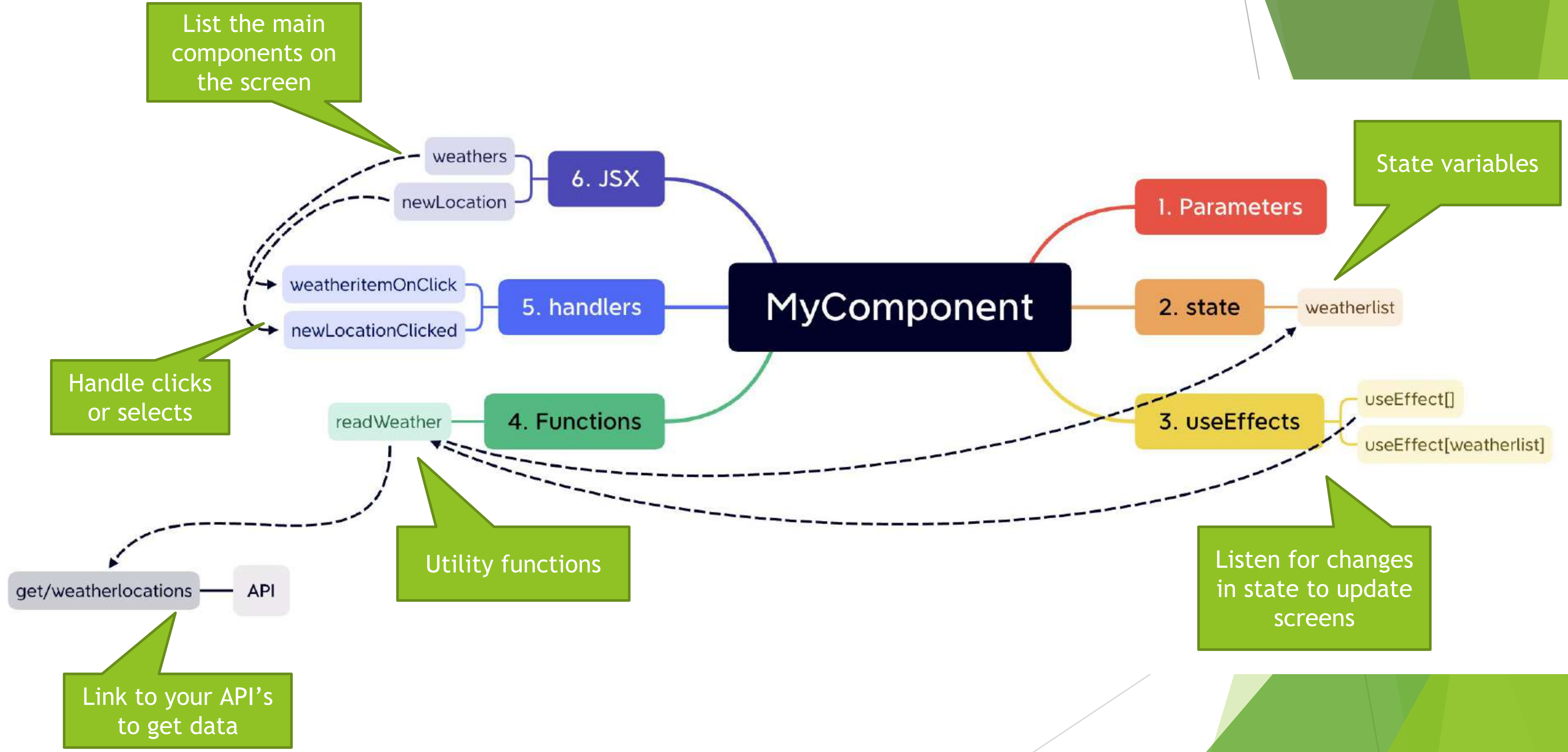
Refresh students on these design documents

Using your designs

Importance of Designing React Components Before Coding

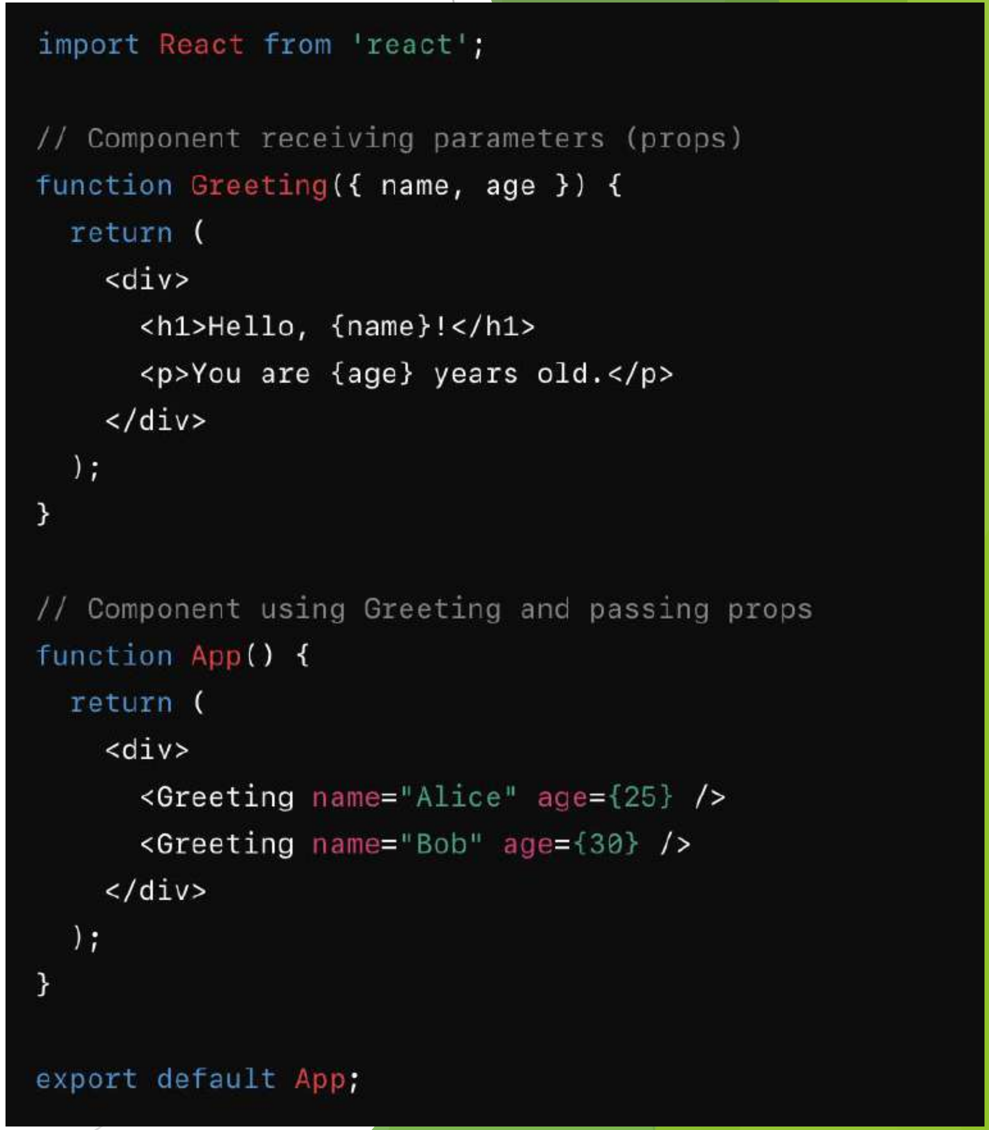
- Clear Structure and Reusability
  - Helps define component hierarchy early
  - Encourages reusability of components
  - Clear understanding of props and state flows
- Simplifies Development
  - Breaks complex UIs into manageable parts
  - Allows for early detection of potential issues
  - Prevents tightly coupled components
- Promotes Collaboration and Consistency
  - Easy for teams to understand component interactions
  - Standardizes patterns for styling and behavior
  - Facilitates shared component libraries across projects
- Saves Time in the Long Run
  - Minimizes refactoring and technical debt
  - Components are easier to test individually
  - Promotes faster iterations and updates

Mindmap to design your screen component



Passing Parameters into React Components

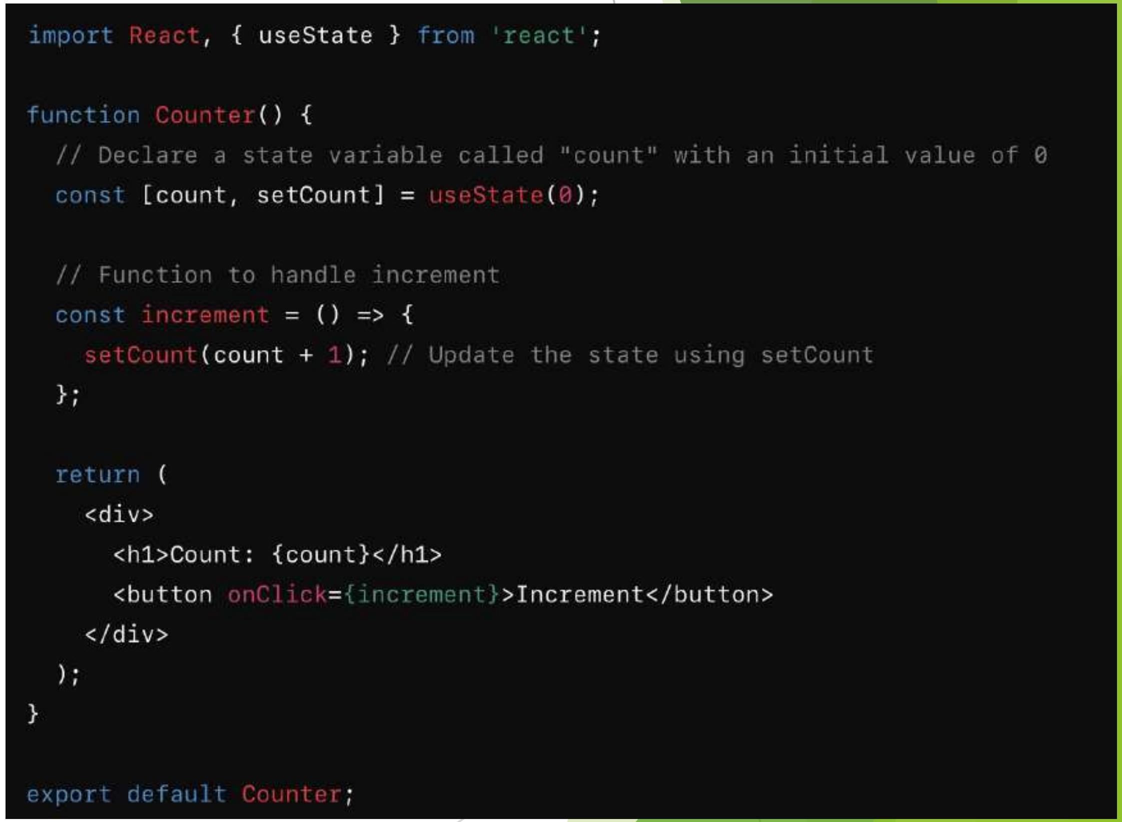
- Props as Parameters
  - React components receive data via props, allowing for dynamic content.
- Destructuring Props
  - Props can be destructured for cleaner and more readable code.
- Passing Multiple Parameters
  - You can pass multiple parameters as individual props to components.
- Default Props
  - Set default values for props if no value is passed by the parent component.
- Prop Types Validation
  - Use PropTypes to enforce the types of props and improve code quality.



```
import React from 'react';

// Component receiving parameters (props)
function Greeting({ name, age }) {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>You are {age} years old.</p>
    </div>
  );
}

// Component using Greeting and passing props
function App() {
  return (
    <div>
      <Greeting name="Alice" age={25} />
      <Greeting name="Bob" age={30} />
    </div>
  );
}

export default App;
```

Understanding useState in React

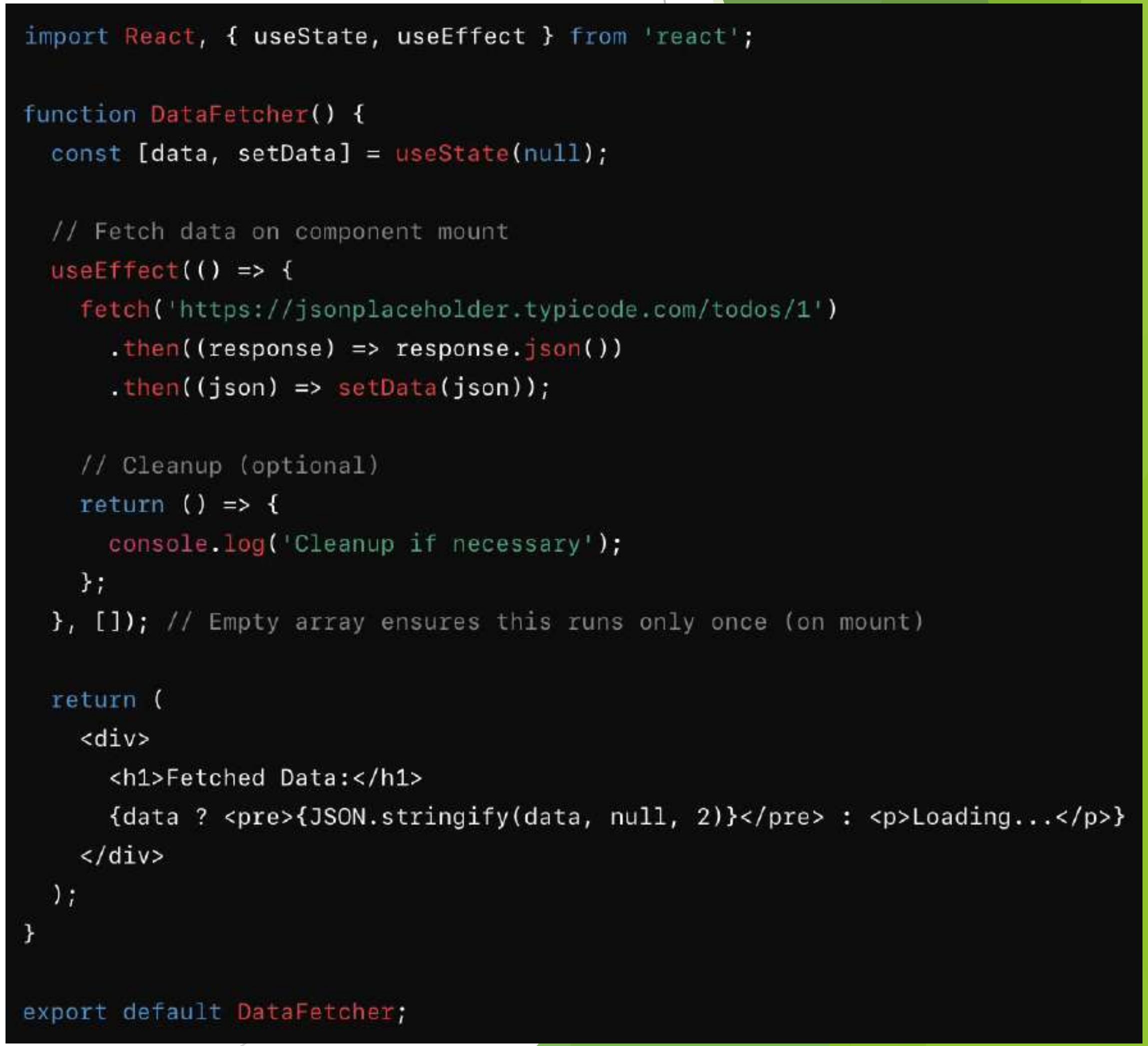
- **State Management in Functional Components:**
  - useState allows functional components to maintain and update local state.
- **Syntax and Usage:**
  - const [state, setState] = useState(initialValue)
  - Provides a state variable and a function to update it.
- **Reactivity and Re-renders:**
  - Updating state triggers re-render, allowing the UI to respond to changes dynamically.



```
import React, { useState } from 'react';

function Counter() {
  // Declare a state variable called "count" with an initial value of 0
  const [count, setCount] = useState(0);

  // Function to handle increment
  const increment = () => {
    setCount(count + 1); // Update the state using setCount
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

Introduction to useEffect in React

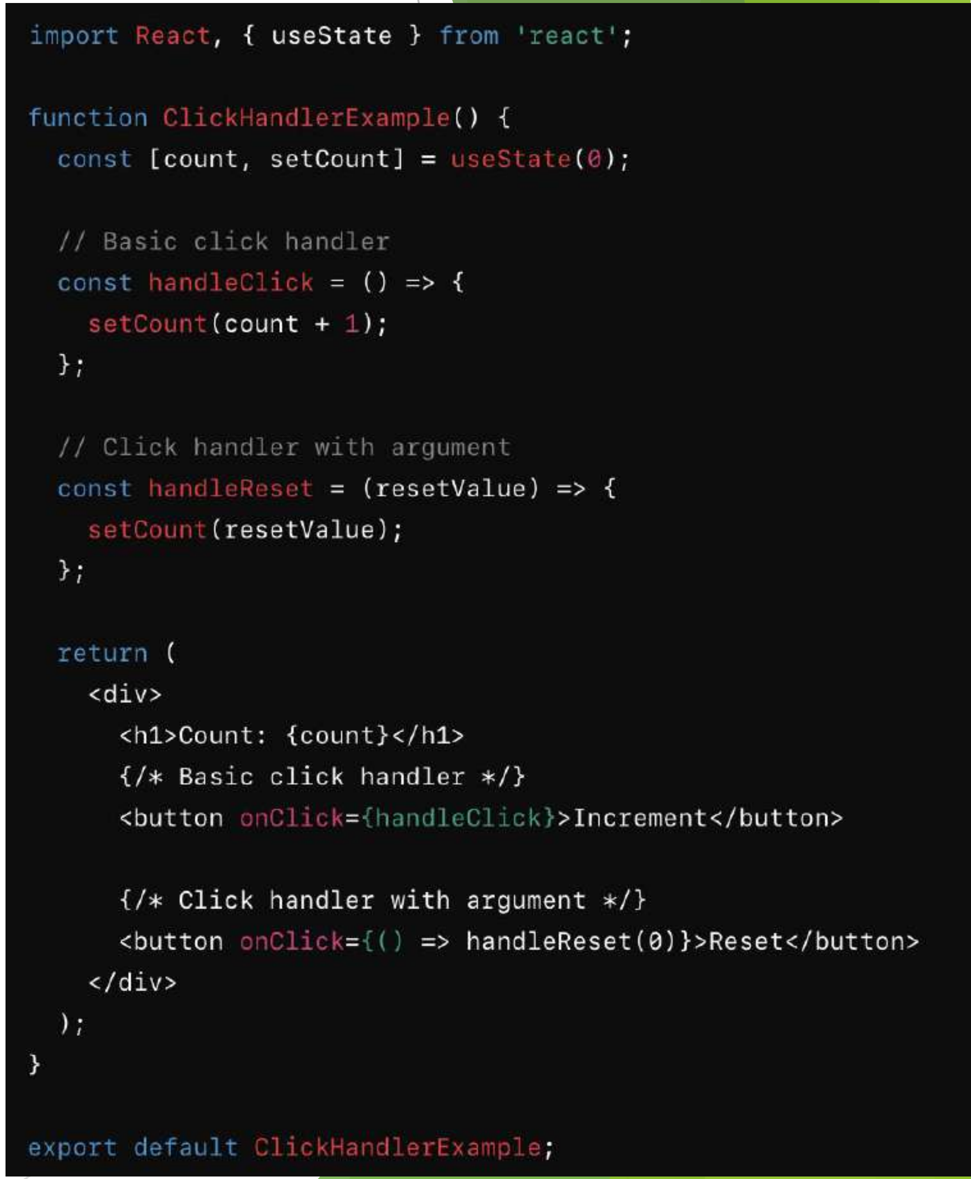
- Side Effects in Functional Components
  - useEffect allows you to perform side effects such as data fetching, DOM manipulation, or subscriptions.
- Runs After Render
  - By default, useEffect runs after every render, ensuring that your side effects are handled after the DOM is updated.
- Dependency Array
  - Controls when useEffect runs. Providing dependencies ensures the effect only runs when specific variables change.
- Cleaning Up Effects
  - You can return a cleanup function from useEffect to avoid memory leaks or unwanted behavior (e.g., removing event listeners).
- Common Use Cases
  - Fetching data from an API, setting up timers, updating document titles, subscribing to streams.



```
import React, { useState, useEffect } from 'react';

function DataFetcher() {
  const [data, setData] = useState(null);

  // Fetch data on component mount
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then((response) => response.json())
      .then((json) => setData(json));

    // Cleanup (optional)
    return () => {
      console.log('Cleanup if necessary');
    };
  }, []); // Empty array ensures this runs only once (on mount)

  return (
    <div>
      <h1>Fetched Data:</h1>
      {data ? <pre>{JSON.stringify(data, null, 2)}</pre> : <p>Loading...</p>}
    </div>
  );
}

export default DataFetcher;
```

Handling Click Events in React

- Inline Functions
  - You can pass an inline function directly to the onClick attribute.
- Event Handling Syntax
  - React uses camelCase for event handlers, e.g., onClick instead of onclick.
- Passing Arguments to Handlers
  - You can pass arguments to click handlers using an inline arrow function.
- Event Object
  - The event object is automatically passed to the handler, providing access to event-specific details like event.target.
- Binding Class Methods
  - In class components, you need to bind methods to this, but functional components handle this more simply.



```jsx
import React, { useState } from 'react';

function ClickHandlerExample() {
  const [count, setCount] = useState(0);

  // Basic click handler
  const handleClick = () => {
    setCount(count + 1);
  };

  // Click handler with argument
  const handleReset = (resetValue) => {
    setCount(resetValue);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      {/* Basic click handler */}
      <button onClick={handleClick}>Increment</button>

      {/* Click handler with argument */}
      <button onClick={() => handleReset(0)}>Reset</button>
    </div>
  );
}

export default ClickHandlerExample;
```

Homework

- Self study
  - https://youtu.be/IYvD9oBCuJI?si=ZiOmKwwQuZCf70SH
  - https://youtu.be/-4XpG5_Lj_o?si=k6C8udkxniDKVnvf
  - https://youtu.be/0XSDAup85SA?si=IP0ObhX6oajuSI9X
  - https://youtu.be/gv9ugDJ1ynU?si=mBJFAtm3OiIf7KRD
  - https://youtu.be/qdCHEUaFhBk?si=Ty9SamXKxdvyPyPZ
  - https://youtu.be/IkMND33x0qQ?si=OVSAb1QFq38OXSK7
- Create design mind maps using xMind for each of your screen components you will need to develop
  - Create one mind map per screen component
  - Ensure it adheres to your
    - System context diagram
    - Flowchart diagrams
    - Screen mockups
  - Upload
    - One mind map per screen component
    - Your system context, flowchart and screen mockups
    - Get approval on your design from your instructor before you begin coding next week
- Complete the weekly quiz